Des Autom Embed Syst (2009) 13: 287–310
DOI 10.1007/s10617-009-9046-2

Runtime deadlock analysis for system level design

Eric Cheung · Xi Chen · Harry Hsieh · Abhijit Davare · Alberto Sangiovanni-Vincentelli · Yosinori Watanabe

Received: 15 February 2008 / Accepted: 13 July 2009 / Published online: 25 July 2009

Abstract In the design of highly complex, heterogeneous and concurrent systems, deadlock detection remains an important issue. In this paper, we systematically analyze the synchronization dependencies in system-level designs. We propose a data structure called the dynamic synchronization dependency graph, which captures the runtime blocking dependencies among concurrent processes. A loop-detection algorithm is then used to detect deadlocks and help designers quickly isolate and identify modeling errors that cause the deadlock problems. We demonstrate our approach through two publicly available system-level modeling languages, SystemC and Metropolis, and two real world design examples, which are complex system-level functional models for video processing.

Keywords Deadlock detection · System-level design · SystemC · Metropolis

E. Cheung (✉) · H. Hsieh
University of California, Riverside, CA 92521, USA
e-mail: chuncheung@cs.ucr.edu

H. Hsieh
e-mail: harry@cs.ucr.edu

X. Chen
SpringSoft, Inc., San Jose, CA 95110, USA
e-mail: xi_chen@springsoft.com

A. Davare · A. Sangiovanni-Vincentelli
University of California, Berkeley, CA 94720, USA

A. Davare
e-mail: davare@eecs.berkeley.edu

A. Sangiovanni-Vincentelli
e-mail: alberto@eecs.berkeley.edu

Y. Watanabe
Cadence Berkeley Laboratories, Berkeley, CA 94704, USA
e-mail: watanabe@cadence.com

1 Introduction

Today's electronic systems become highly complex, highly heterogeneous, and highly concurrent. The platform-based system-level design methodology is increasingly being adopted as the primary method to deal with the complexity of these modern systems. It becomes an important part of the design methodology in the electronic design industry. The earlier in the design process the designers can locate a problem, the less time and resources will be needed to fix it. When a design is synthesized from one level to a more detailed level, such as from the behavioral level to the RTL level or from the RTL level to the gate level, functional verification and debugging become much more complex and expensive. Furthermore, problems that occur at higher abstraction levels are often hidden in the implementation details at the lower levels, making design verification even more difficult and time consuming. Decreasing time-to-market and increasing design cost are pushing design, analysis, and verification above the RTL level. System-level verification approach is necessary to catch any violation of communication constraints or synchronization problems caused by design errors. We focus on system-level designs where the processes can represent software and hardware. The processes are modeled in the transaction level, which means process dependencies will be explicitly shown in the simulator by asynchronous communication. Computation and communication are abstracted in the models.

Even with careful methodological guidance, it is still possible to introduce unintended and undesirable behaviors into function specifications, high level architecture models or function-architecture mappings. The complex interactions of the communication, computation, and data-path control components in a system-level design make it very vulnerable to synchronization errors. Foremost among these are *deadlocks*, *livelocks* and *starvations*. Many system designers encounter deadlocks on industrial examples on a daily basis. Many of these deadlock problems come in the context of architectural mappings and coordination interactions. In this context, deadlocks are very confusing due to other running processes and communication resources like mutex. To manually determine a deadlock, the designers would have to integrate additional debugging messages into the models. By the time an error or an assertion is shown in the simulation, the deadlock may have occurred for some period of time, and the designers would need to use a debugger to step through the long simulation until the originating point of the deadlock is observed. Such procedures takes a lot of experience of the designers and would take hours or even days.

In our MPEG-2 Decoder and Picture-in-Picture Video Processing System, designs of the applications are very complex and design components are modeled as processes running concurrently and asynchronously. It is very difficult to analyze the interactions between the processes. In addition, simulation of such systems is expensive and takes long periods of time, and we would like to catch the problems in the designs earlier in the simulation to avoid long simulation traces and unexpected interactions between processes after the problems occur. Being semantic in nature, the complete and precise characterizations of the systems require formal analysis or verification. Although it is able to consider different input vectors, it can only be done for the simplest of designs due to the state space explosion problem. In this work, We look for a practical solution to deal with these design problems in realistic and complex system designs. A simulation based analysis methodology is proposed for the detection and elimination of these "semantic errors". Designers are responsible for coming up with simulation vectors and scenarios that are important and may lead to undesirable behaviors such as deadlocks. Our approach automatically analyzes the simulation status and reports deadlocks once they occur.

In this paper, we analyze and identify the deadlock problems in system-level simulation. We propose a data structure to be built during simulation that reflects the runtime

blocking dependencies among concurrent processes. We follow up with an associated deadlock detection algorithm to monitor the simulation inside the simulation kernel engine. The goal of the algorithm is to help the designers identify the components and synchronization constructs that cause the deadlock problems and provide an error trace and a history of dependency snapshots that show how the system arrives at the deadlock state. Our method of extending the simulation kernel is generic in nature and can be easily fit into any simulation platforms. We use two publicly available system-level modeling languages and simulators, *SystemC* and *Metropolis* to demonstrate and validate our approach. We also use two real world designs, an MPEG-2 Decoder and a Picture-in-Picture Video Processing System, to demonstrate the usefulness and effectiveness of the deadlock analysis.

In the next section, we discuss the related works. In Sect. 3, we introduce synchronizations normally used in system-level simulation and show various deadlock situations that can be caused by the synchronizations. We also illustrate the data structure of the dynamic synchronization dependency graph and the algorithms for deadlock detection. In Sect. 4, we present the implementations in SystemC and Metropolis platforms. We also present the evaluations of the case studies in each of the platforms. We conclude the paper in Sect. 5 and discuss some future directions in Sect. 6.

2 Related works

Deadlock detection and resolution techniques have already been extensively studied in the areas of operating systems and database systems [1–9]. In those domains, deadlock prevention is possible if particular resource allocation policies are applied. Deadlock avoidance is used as a part of scheduling algorithms to choose at least one possible execution path where no deadlock will occur. A resource allocation graph or state graph is usually used to analyze and identify deadlock situations for deadlock detection. Though it is possible to incorporate these techniques in system designs to eliminate deadlocks, they are not general enough to apply to arbitrary designs due to the design flexibility required by today's platform-based embedded systems. Our deadlock analysis mechanism is integrated into the design framework (rather than the designs) to help designers analyze design errors while allowing full design flexibility.

In concurrent software, various formal verification techniques are employed to exhaustively search for deadlock situations in concurrent protocols [10–14].[1] In essence, synchronization protocols at a high level of abstraction, either extracted from the designs or defined *a priori*, are formally verified. However, modern synchronization structures are becoming too complex and their analysis suffers from the state explosion limitation. Our approach is based on simulation, so it can handle real complex system designs.

In simulation verification, assertions that are based on temporal logics can be used to check for safety properties in a certain period of execution.[2] However, temporal assertions have to be designed according to particular applications. They are usually used to guard the overall behaviors of the systems, and not suitable for identifying the causes of those undesirable behaviors due to their "trace checking" natures. A general deadlock detection mechanism is proposed in [15] for discrete event simulation models. However, no implementation on real simulation models are discussed in the literature.

[1] Verisoft. http://cm.bell-labs.com/who/god/verisoft/.

[2] http://www.eda.org/vfv, 2003.

In the emerging simulation environment for heterogeneous system-level designs, an effective and efficient deadlock analysis tool that can be tightly integrated into the design methodology is needed, which is the main focus of this paper. Our mechanism to incrementally search for deadlock during simulation is entirely novel and represent new directions in simulation verification of system-level designs.

3 Synchronization dependency and deadlock analysis

In this section, we introduce a deadlock analysis methodology for system-level designs. We propose a data structure called the *dynamic synchronization dependency graph (DSDG)* for deadlock analysis. Once the synchronization dependencies are captured by the graph, an algorithm can be used to detect deadlock situations.

In system-level simulation, high level processes run synchronously in a discrete event simulator. Processes do not perform operations independently, they interact with the other processes to perform the desired functionality. When the interactions between processes are incorrectly designed or implemented, a set of processes may wait for each other and form a cyclic dependency. When this happens, the set of processes would not have any chance to continue execution.

Our deadlock analysis methodology is illustrated in Fig. 1. By integrating deadlock analysis into a simulation environment for system-level designs, designers can efficiently analyze complex concurrent systems with simulation and quickly identify design problems that may cause deadlocks. The task of design analysis becomes much easier with the help of runtime synchronization information combined with the regular simulation traces and the static network structures. They can be used to guide the designers to revise the design to eliminate the problems or modify the simulation vectors to explore different execution paths for other design errors. This methodology allows full design flexibility and is able to handle

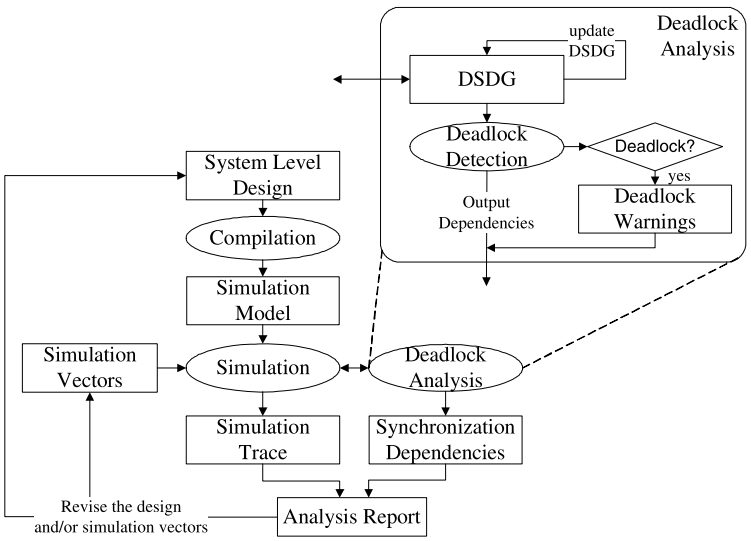

Fig. 1 Deadlock analysis methodology

large models. The details of the deadlock analysis mechanism will be discussed in the rest of this section.

We propose a data structure called the dynamic synchronization dependency graph (DSDG), which captures the runtime blocking dependencies. A loop-detection algorithm is then used to detect deadlocks and help designers quickly isolate and identify modeling errors that cause the deadlock problems.

Our methodology does not consider "livelocks", where the processes are constantly executing without progressing. Since the definition of "progress" is application-dependent, it is impossible for the simulator itself to distinguish between a livelock and a normal execution loop, such as a clock generation. Our methodology also does not consider "blocking chains", where a process suspends itself and therefore processes that depend for it have to wait. It happens very often in normal execution and is not considered a deadlock. Our methodology detects deadlocks where each of the processes waits for other processes involved in the deadlocks, hence stop all the involved processes to executed any further.

3.1 Dynamic synchronization dependency graph

Definition 1 A deadlock is a situation where two or more processes are blocked in execution while each is waiting for others and there is no possibility of continuing execution.

Definition 2 A dynamic synchronization dependency graph (DSDG) is a directed graph $S = (V, E)$. V is a set of vertices representing processes in the network and dynamic dependency nodes. E is a set of directed edges between vertices indicating dynamic synchronization dependencies.

In a DSDG, each process in the network is represented by a process vertex. Other dependency vertices and edges are added and deleted dynamically as dependencies between processes change in the execution. A dynamic dependency nodes are additional vertices that are used to allow more complex dependencies such as AND- and OR-dependencies that will be introduced later. Dynamic synchronization dependencies are directed edges that are used to represent that processes are blocked by system-level synchronization statements. System-level synchronization statements are normally used to synchronize concurrent running processes. If a process is blocked by a synchronization statement, it has dependencies on the processes that may change the evaluation of the synchronization statement sometime later.

In addition to a single dependency, in system-level designs a process can also have a more complex dependency based on system-level modeling language dependent constructs. In a DSDG, a process vertex can only have one outgoing edge to another process vertex or a dynamic dependency node. A complex dependency is represented by a combination of dynamic dependency nodes (vertices) and dynamic synchronization dependencies (directed edges). AND-dependencies and OR-dependencies are examples of common complex synchronization constructs that exist in many system-level languages. An AND-dependency requires a process to be blocked until *all* the conditions become satisfied. If a process waits for an AND-dependency of A, B and C, the process can continue in the earliest time after *all* A, B and C are satisfied. An OR-dependency indicates that as long as *one* of the conditions becomes valid, a process can be released from the dependency. If a process waits for an OR-dependency of A, B and C, the process can continue in the earliest time when one of the A, B and C is satisfied. AND-dependencies and OR-dependencies are represented in the DSDG

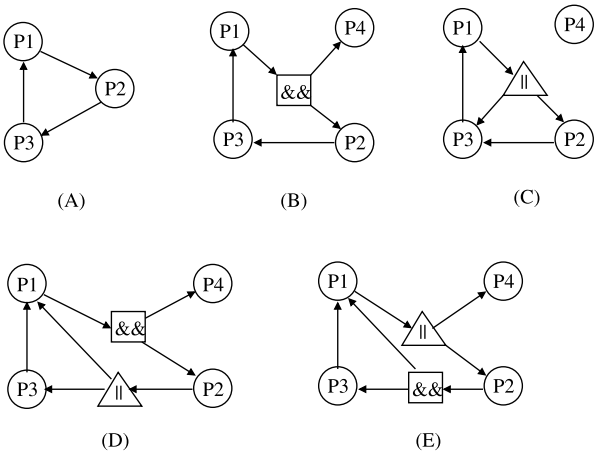

Fig. 2 DSDG Examples (**A**) deadlock with simple cyclic dependency. (**B**) deadlock with AND-dependency. (**C**) deadlock with OR-dependency. (**D**) deadlock with both AND- and OR- dependencies. (**E**) no deadlock

Algorithm 1 Main procedure to build and update a DSDG

```
1:  procedure UPDATE_DSDG()
2:  for each process p_i in the system do
3:      if p_i is unblocked by one or more synchronization constructs then
4:          remove all the dependency vertices and edges from p_i caused by those synchronization
            constructs;
5:      end if
6:      if p_i is blocked by one or more synchronization constructs then
7:          UPDATE_PROCESS(p_i);
8:      end if
9:  end for
10: end procedure
```

examples shown in Fig. 2 as added dynamic dependency nodes and dynamic synchronization dependencies. System-level modeling languages usually have language dependent synchronization constructs. In SystemC, synchronization is based on events. In Metropolis, resource managers and eval-dependencies could be used to synchronize processes. Examples of synchronization constructs for SystemC and Metropolis are shown in Sects. 4.1 and 4.2, respectively.

A DSDG is automatically built and updated during simulation. It describes the status of dependencies among all the concurrent processes in the system at a particular execution state. If a process is actively running, there is no *outgoing* edge from it in the graph. When it is blocked or released, dependency vertices and edges are added to or deleted from the graph dynamically at the time it happens (see Algorithm 1 to 3). Initially, the DSDG only includes all the process vertices V, and edges E is set to ϕ. During the simulation, UPDATE_DSDG() is called to update the DSDG every time the synchronization dependencies of the system are changed. UPDATE_PROCESS() is called to update the DSDG for each blocked process. CONNECT() connects newly added vertices with directed edges.

Here we show some examples of DSDG and some of the deadlock situations that can be detected in DSDG.

1. A DSDG shows a simple cyclic dependency among three processes, such as one that would be shown in dining philosophers, has been shown in Fig. 2A.

Algorithm 2 Procedure to handle a blocked process

```
1: procedure UPDATE_PROCESS(p_x)
2: for each synchronization construct that blocks p_x do
3:    if p_x is blocked by a single dependency to p_y then
4:       CONNECT(p_x, φ, {p_y});
5:    else if p_x is blocked by a synchronization construct that requires its waiting for any of
      processes p_1, p_2, ..., and p_n then
6:       add an OR-dependency vertex o_x;
7:       CONNECT(p_x, o_x, {p_i : i ∈ [1, n]});
8:    else if p_x is blocked by a synchronization construct that requires its waiting for all of
      processes p_1, p_2, ..., and p_n then
9:       add an AND-dependency vertex a_x;
10:      CONNECT(p_x, a_x, {p_i : i ∈ [1, n]});
11:   end if
12: end for
13: end procedure
```

Algorithm 3 Procedure to connect newly added vertices

```
1: procedure CONNECT(src, mid, {dest_i : i ∈ [1, n]})
2: if mid = φ then
3:    add an edge from src to dest_1;
4: else
5:    add an edge from src to mid;
6:    for i := 1 to n do
7:       add an edge from mid to dest_i;
8:    end for
9: end if
10: end procedure
```

2. A DSDG shows a deadlock situation with an AND-dependency has been shown in Fig. 2B. P1 waits for both P2 *and* P4, where P2 is blocked by P3 which in turn is blocked by P1. Therefore, P1, P2 and P3 form a deadlock and cannot continue.
3. A DSDG shows a deadlock situation with an OR-dependency has been shown in Fig. 2C. P1 waits for P2 *or* P3, P2 waits for P3, and P3 waits for P1. Therefore, P1, P2 and P3 form a deadlock and cannot continue.
4. A DSDG shows a deadlock situation with both an OR-dependency and an AND-dependency has been shown in Fig. 2D. P2 is waiting for either P1 and P3, and both of them are blocked. P3 is waiting for P1, and P1 is waiting for both P2 *and* P4. Since P1 cannot continue unless P2 continues, P2 cannot continue unless one of P1 and P3 continue, and P3 cannot continue unless P1 continues, there is a deadlock formed in P1, P2 and P3.
5. A DSDG shows a situation with both an OR-dependency and an AND-dependency but *without* deadlock has been shown in Fig. 2E. P1 is waiting for either P2 or P4. Since P4 is not blocked, it can unblock P1 (OR-dependency) and eventually unblock P3 then P2. Hence there is *no* deadlock.

We implement our DSDG by adjacency list representation with an array of lists of vertices. Each vertex can be labeled to indicate the exact location in the source code that it is

corresponding to. This information can be made available for the designers to help identify the problems quickly.

3.2 Deadlock detection algorithm

A deadlock happens when two or more processes are waiting for each other and there is no possibility of continue execution. Without AND- and OR-dependencies, a deadlock is a cyclic dependency in the DSDG. For an AND-dependency to be involved in the deadlock, only one vertex from its outgoing edges has to be inside the set of deadlock processes. For an OR-dependency, all vertices from its outgoing edges has to be inside the set of deadlock processes. Therefore, a cyclic dependency must happen for every outgoing edge from an OR-dependency in order for it to be involved in a deadlock. Such dependency analysis provides *exact* deadlock detection without any false positives or false negatives.

Every time a process suspends in the simulation, the dynamic synchronization dependency graph is updated and the deadlock detection algorithm is applied. The algorithm is shown in Algorithm 4. The DSDG and the process blocked are two inputs to the algorithm.

Algorithm 4 Deadlock detection.

```
1:  procedure DETECT_DEADLOCK(S, P)
2:  search for simple cycles in S from process vertices in P;
3:  let 𝕃 = {L_i = (V_i, E_i)} be the set of all these simple cycles;
4:  if no new simple cycle is added to 𝕃 then
5:      return NO_DEADLOCK;
6:  end if
7:  for each L_i ∈ 𝕃 do
8:      if L_i is already marked then
9:          continue;
10:     end if
11:     mark L_i;
12:     if each vertex in V_i is either a process or AND-dependency then
13:         the processes in L_i are deadlocked, return;
14:     else
15:         D := {OR-dependency vertices in V_i that have two or more outgoing edges};
16:         𝕃' := {L_i};
17:         repeat
18:             find unmarked cycles in 𝕃 that contains vertices in D;
19:             mark all these cycles;
20:             D := D ∪ {OR-dependency vertices with two or more outgoing edges in these
                cycles};
21:             𝕃' := 𝕃' ∪ {these cycles};
22:         until 𝕃' becomes stable
23:         if ∃ vertex in D that has an outgoing edge ∉ 𝕃' then
24:             continue;
25:         end if
26:         the processes in 𝕃' are deadlocked, return;
27:     end if
28: end for
29: return NO_DEADLOCK;
30: end procedure
```

The algorithm searches for cyclic dependencies and determines deadlocked processes. There are two major steps in the algorithm. First step is to find all the simple cycles in the graph starting from the blocked process. By definition, a simple cycle in a directed graph is a cycle that does not pass through a vertex more than once. To detect all the simple cycles that involves a particular vertex is to do a depth-first-search (DFS) search from that vertex. Then to detect all the simple cycles in the graph, do a DFS for each vertex. And the next step is to traverse all the simple cycles and to determine any cyclic dependencies. The second step contains two cases. First, if a cycle contains only single dependency vertices and AND-dependency vertices then it is a deadlock (lines 12–13); second, if a cycle contains a mixture including OR-dependency vertices, it is a deadlock if only all the outgoing branches of the OR-dependency vertices contains in some cycles (lines 14–26).

With a mixture of single, AND-, and OR-dependencies, only OR-dependencies need special attention because one single outgoing edge in the OR-dependency vertex is able to unblock the process, which makes a cyclic dependency an insufficient condition for a deadlock if an OR-dependency is involved. For a single or an AND-dependency, the process waits for all outgoing edges which make them the same as resource-allocation graph and no special attention is needed. If a cycle contains OR-dependencies, all the outgoing branches of the OR-dependency vertices must be contained in some cycles for them to be involved in a deadlock. These steps repeat if those cycles also contain OR-dependencies. This is done until all outgoing branches of all OR-dependency vertices involved have been considered (lines 17–22 in Algorithm 4). If all outgoing edges in all OR-dependency vertices involved are contained in cycles, there is a deadlock (lines 23–26).

Given a dynamic synchronization dependency graph $S = (V, E)$ and a set of processes that are blocked from running P, we use the algorithm to detect deadlock situations. Generally, the algorithm traverses the graph, searches for cyclic dependencies, and determines deadlocked processes. The algorithm not only decides if there is any deadlock but also identifies all the processes and synchronization constructs that are involved in the deadlock situations. In the worst case, the first step of the algorithm is to find all the simple cycles in the graph. The DFS search from each vertex has complexity of $O(|V| + |E|)$. So detecting all the simple cycles in the graph has the complexity of $O(|V| \cdot (|V| + |E|))$ assuming that the adjacency-list representation is used for the graph. The rest of the algorithm will traverse all the simple cycles at most twice with a complexity of $O(|V|)$. If a simple cycle only contains process vertices and AND-dependency vertices, then it is a deadlock. If a simple cycle also contains OR-dependency vertices, there is a deadlock only if other edges from these OR-dependency vertices all lead to cycles. Therefore, the complexity of the algorithm is $O(|V| \cdot (|V| + |E|))$. $|V|$ and $|E|$ are the numbers of vertices and edges, which are determined by the number of process instances and dependencies in a system in a DSDG. In reality, the overhead for deadlock detection is relatively low and constant throughout the simulation. A DSDG is normally a sparse graph that does not reassemble the worst case. The detection returns immediately if the newly added edge does not introduce any new simple cycle.

4 Implementations

The dynamic synchronization dependency graph and deadlock detection algorithm have been implemented in the SystemC and Metropolis simulators. During the simulation of a design, a DSDG is built and updated as the dependencies of the system changes, i.e. as one or more processes in the system are blocked from running or released from blocking. Whenever one or more processes are blocked from running, the deadlock detection algorithm is

invoked to search the DSDG for any deadlock situation. Once a deadlock is detected in the simulation, the history of DSDG updates provides a trace that shows how the system execution goes into the deadlock. Due to the incremental nature of the DSDG updates and deadlock detection algorithms, this simulation monitoring mechanism will not introduce significant overhead to the regular simulation. The overhead for each deadlock detection only depends on the complexity of the DSDG, and the DSDG normally remains a sparse graph.

4.1 SystemC

SystemC [16–19] enables system-level modeling of complex systems that can be implemented with software, hardware, or combination of the two.[3] One of the challenges in providing a system-level design framework is that there is a wide range of models of computation, abstraction levels, and design methodologies used in the design flows. To address this challenge, SystemC provides a core language that introduces a new set of constructs for generalized modeling of communication and synchronization at the very high abstraction level. On top of this language foundation the designers can then add more specific models of computation, design libraries, modeling guidelines, and design methodologies that are required for system-level designs. SystemC allows executable specifications of system designs based on C++ programming language extended with constructs for describing concurrency, timing, and reactivity modeling of parallel systems containing both software and hardware. This is the reason why SystemC gains popularity as a de facto standard for modeling SoC designs.

SystemC has gained popularity as a system-level modeling language. Efficient and accurate simulation of the SystemC designs have become increasingly important. Efficient deadlock detection in SystemC is accomplished by extending the SystemC simulation kernel to build the DSDG incrementally at runtime, and then applying the deadlock detection algorithm to the graph.

4.1.1 Synchronization language

SystemC models are simulated through a discrete event simulation kernel that schedules events at runtime. SystemC includes a set of language constructs such as channels, interfaces, and events, as well as modeling primitives such as queues, semaphores, memories, and buses to provide support for system-level modeling. The simple and flexible synchronization capability provided by events and wait() or next_trigger() methods allow a broad range of channels specified in the design.

In SystemC, processes are used to describe the functionality and specify concurrency of a system. Processes are contained in modules and they access external channel interfaces through ports. A process can be SC_METHOD, SC_THREAD or SC_CTHREAD. An SC_METHOD process is triggered on events and runs to completion once triggered. It supports synchronization through the next_trigger() method that returns immediately without passing control to another process. An SC_THREAD process can be suspended during execution and resumed at a later stage. It executes once during simulation and the wait() method is mostly used for synchronization purposes. An SC_CTHREAD process is usually sensitive to a single clock but can be suspended by an explicit wait for events. For a

[3] SystemC version2.0 user's guide, Copyright 1996–2002.

process to be explicitly blocked for deadlock detection analysis, the process needs to wait for events either by a static sensitivity (SC_METHOD or SC_THREAD) or a dynamic sensitivity (SC_METHOD, SC_CTHREAD or SC_THREAD).

Events are primitive behavior triggers. A process can suspend on or be sensitive to one or more events. The sensitivity of a process defines when this process will be resumed or activated. Whenever one of the corresponding events is triggered, the process is resumed or activated. If the sensitivity of the process is declared during elaboration and cannot be changed once simulation has started, then it is called static sensitivity. In some instance we want a process to be sensitive to a specific event or a specific collection of events that may change during simulation. This is called dynamic sensitivity and is done by using the next_trigger(), wait() and notify() method. The next_trigger() method is called in the method process to set the dynamic sensitivity of the process for the next occasion. The calling method process is triggered when one or all of the specified events is notified. The wait() method is called anywhere in the thread of execution of a thread process. When it is called, the specified events temporarily overrule the sensitivity list, and the calling thread process suspends. When one or all of the specified events is notified, the waiting thread process is resumed. If a process is sensitive to a clock edge or waits for time, the process is not considered blocked by other processes in the simulation, since the process can be unblocked by a clock edge or time. Following is a list of different forms of next_trigger() and wait() method to wait for events that are supported by SystemC.

1. next_trigger($e1$): This next_trigger method set the next trigger of the current process to wait until the event e1 is notified.
2. next_trigger($e1|e2|e3$): This next_trigger method set the next trigger of the current process to wait until any of the events e1, e2, or e3 is triggered.
3. next_trigger($e1\&e2\&e3$): This next_trigger method set the next trigger of the current process to wait until all of the events e1, e2, and e3 are triggered.
4. wait($e1$): This wait method suspends execution of the current process until the event e1 is notified.
5. wait($e1|e2|e3$): This wait method holds the current thread process until any of the events e1, e2, or e3 is triggered.
6. wait($e1\&e2\&e3$): This wait method holds the execution of current thread process until all of the events e1, e2, and e3 are triggered.

The semantics of the next_trigger() or wait() method with one or more event arguments is that the method process is triggered or the thread method returns (i.e. the thread of execution is resumed) either when at least one of the events is notified or when all the events are notified. A mixture of j operator and & operator is not supported by SystemC. SystemC provides a set of channels and corresponding events. The event type sc_event supports user defined channel types. The different functionality that sc_event provides are constructor, notify() and cancel(). Processes sensitive to the next_trigger() or wait() event will trigger or resume after the notification of the event. Thread process waiting for resources using mutex and the semaphore will resume after unlock() and post() respectively, which internally uses sc_event notifications. While basic SystemC does not provide specific real-time scheduling constructs. The designers can still model controller/scheduler explicitly in SystemC by writing a processor that explicitly schedule other processes through wait/notify. Admittedly, this is modeling only for simulation and not likely to be part of the final implementation. It can still be used to catch synchronization errors involving scheduling.

Deadlock with Wait()

```
SC_MODULE(deadlock1){
 sc_event e1, e2, e3;
 void process1(){
   wait(e1);
   e3.notify();}
 void process2(){
   wait(e2);
   e1.notify();}
 void process3(){
   wait(e3);
   e2.notify();}
 SC_CTOR(deadlock1){
   SC_THREAD(process1);
   SC_THREAD(process2);
   SC_THREAD(process3);}
}
```

(A)

Deadlock with Wait(|)

```
SC_MODULE(deadlock2){
  sc_event e1, e2, e3;
  void process1(){
    wait(e2|e3);
    e1.notify();}
  void process2(){
    wait(e3);
    e2.notify();}
  void process3(){
    wait(e1);
    e3.notify();}
  SC_CTOR(deadlock2){
    SC_THREAD(process1);
    SC_THREAD(process2);
    SC_THREAD(process3);}
}
```

(B)

Fig. 3 Deadlock in SystemC

Deadlock with Wait(&)

```
SC_MODULE(deadlock3){
 sc_event e1, e2, e3;
 void process1(){
   wait(e2&e3);
   e1.notify();}
 void process2(){
   wait(e3);
   e2.notify();}
 void process3(){
   wait(e1|e2);
   e3.notify();}
 SC_CTOR(deadlock3){
   SC_THREAD(process1);
   SC_THREAD(process2);
   SC_THREAD(process3);}
}
```

(A)

Deadlock with lock()

```
SC_MODULE(deadlock4){
 sc_mutex m1;
 sc_event e1, e2;
 void process1(){
   m1.lock();
   wait(e1);
   m1.unlock();}
 void process2(){
   wait(e2);
   e1.notify();}
 void process3(){
   if(m1.trylock()){
     /*Do Something*/
     e2.notify();}}
 SC_CTOR(deadlock4){
   SC_THREAD(process1);
   SC_THREAD(process2);
   SC_THREAD(process3);}
}
```

(B)

Fig. 4 More deadlock in SystemC

4.1.2 Deadlock in SystemC

Given the events, sensitivity lists and channel dependencies, the following situations are examples of deadlock in SystemC. Here we use thread processes to enhance the readability of the examples, but the same principle also applies for method processes.

1. A deadlock having simple cyclic dependency due to wait on event. It is shown in Fig. 3A.
2. A deadlock situation having dependencies with wait on event and wait on an OR-sensitivity. It is shown in Fig. 3B.
3. A deadlock situation having dependencies with wait on event, wait on an OR-sensitivity and wait on an AND-sensitivity. It is shown in Fig. 4A.
4. A deadlock situation where processes are suspended due to mutex sharing, which uses sc_event internally. It is shown in the Fig. 4B.

SystemC communication constructs, such as sc_fifo, sc_mutex, and sc_semaphore are made out of basic wait and notify so can be captured in the same way. Our approach, however, will not capture so-called implicit deadlocks, such as those caused by busy waiting loops for changes in shared memory. We believe that it is good programming practice, and indeed is promoted by Transaction-Level Modeling for performance reasons, to always model the communication between processes by explicit communication constructs, such as using sc_event.

4.1.3 SystemC specific implementation

The DSDG and deadlock detection algorithm have been implemented in the SystemC simulator. In SystemC, processes do not directly waiting for another processes, instead they wait for events that would be notified by other processes. Hence in the DSDG, we represent both processes and events as vertices. When process waits for a event, an outgoing edge will be added to the DSDG toward the event vertex the process is waiting for. And event vertices have outgoing edges to the processes that would notify the event. The deadlock detection algorithm is called every time a process suspends. We have implemented the algorithm on an incremental basis so that each time we call the algorithm, we just search the cycles that has been affected by the newly added edges. Due to this, there is no significant overhead to the original simulation, as will be shown by experimental result in the next section.

One issue arises during the building of the DSDG graph in SystemC. Since an event can be notified by one of the multiple processes, any of these processes may notify that event. As a consequence, an OR-dependency is built into the DSDG graph that the waiting process is dependent on an *OR* of the notifying processes. If all of the notifying processes are blocked, then the waiting process can participate in a deadlock. Practically, we first build up a database of the notifying processes for any particular wait through simple searching through the syntax tree. This database remains unchanged throughout simulation.

4.1.4 Dining philosopher example

Figure 5A shows the dining philosopher problem with 5 philosophers and 5 chop sticks. This has been implemented in SystemC considering each philosopher as a process and each chop stick as a mutex and each stage of the philosopher such as thinking, waiting for left chop stick, waiting for right chop stick, eating for a while and unlocking both chop sticks as internal process states. When a philosopher tries to pick up a chop stick that is already picked up by another philosopher, the philosopher is blocked and wait for the event that the chop stick is put down. A deadlock occurs if all philosophers pick up the left chop sticks and wait

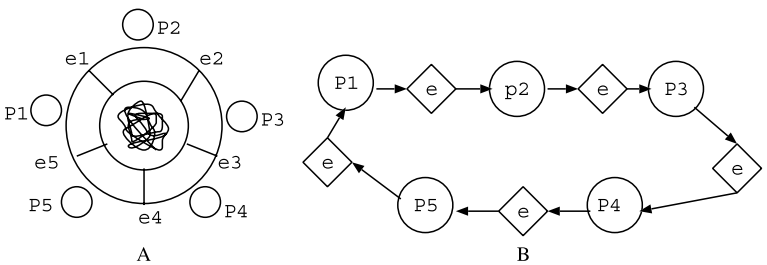

Fig. 5 (**A**) Dining philosopher problem. (**B**) DSDG

Table 1 Run time analysis of dining philosopher problem

With deadlock (no monitor)			With deadlock (monitor)		
#Steps	#Clk Edges	Time (sec)	#Clk Edges	Time (sec)	Overhead (%)
1000	100	0.05	112	0.05	2.00
5000	250	0.13	251	0.14	3.84
10000	557	0.28	565	0.29	4.30
20000	1225	0.61	1235	0.64	4.50
50000	2423	1.25	2449	1.31	4.80
100000	5009	2.51	4910	2.64	4.81
250000	12555	6.28	Deadlock found at # 4910		
1000000	50269	25.13			

for the right chop sticks. We simulated this design and found that the simulator discovered the deadlock, also gave details of the processes involved in the deadlock. The dynamic synchronization dependency graph has been shown in the Fig. 5B and the simulation results are shown in Table 1.

In Table 1 the column *with deadlock (No monitor)* presents the run time of the regular SystemC simulation model and the column *with deadlock (monitor)* presents the run time of our proposed model. The simulation resolution time has been set at 50 μs and the simulation has been set to simulate forever so that each philosopher will be continuing eating and thinking. We can see that even the deadlock occurs at 4910 clock cycle, the simulate() and crunch() inside the simulation engine run forever watching for any process to be ready. But with our monitoring mechanism we are able to detect the deadlock early as soon as it occurs. We raise a deadlock flag and also we report the dependencies and processes involved in the deadlock. The overhead on the SystemC simulator due to the additional procedure calls is below 5%. This is because the work to do for updating the graph and searching the cyclic dependencies is done incrementally.

4.1.5 Case study: process network MPEG-2 decoder

We demonstrate our implementation in SystemC through a real world design example, an MPEG-2 Decoder [20]. We use an Intel Xeon 1.4 GHz processor with 512 MB of RAM for our experiments. We do a run time comparison between the regular simulation model and the simulation model with our proposed enhancements. The overhead on the overall simulation is shown to be insignificant.

The MPEG-2 Decoder system-level design is shown in Fig. 6. The design is modeled as a Kahn Process Network [21]. Processes are connected and communicate through unbounded FIFO queues. In the design, controller processes control the dataflow of the MPEG-2 video stream. When the video stream enters the decoder, it is first parsed into MPEG-2 frames. Each frame is decoded through the inverse scan (IS), inverse quantization (IQ) and inverse discrete cosine transforms (IDCT) process pipeline. Prediction processes perform motion compensation and prediction on the frames. Output processes combine the raw video streams and output the stream from the decoder. We implement the design in SystemC and limit the sizes of the FIFO queues. Each of the process is implemented as a thread process and FIFO queue is implemented as a sc_fifo with finite size. Reading from a FIFO queue when empty is blocking and the calling process is suspended to wait for the event that another process writes to the FIFO queue. Similarly, writing to a FIFO queue when full is

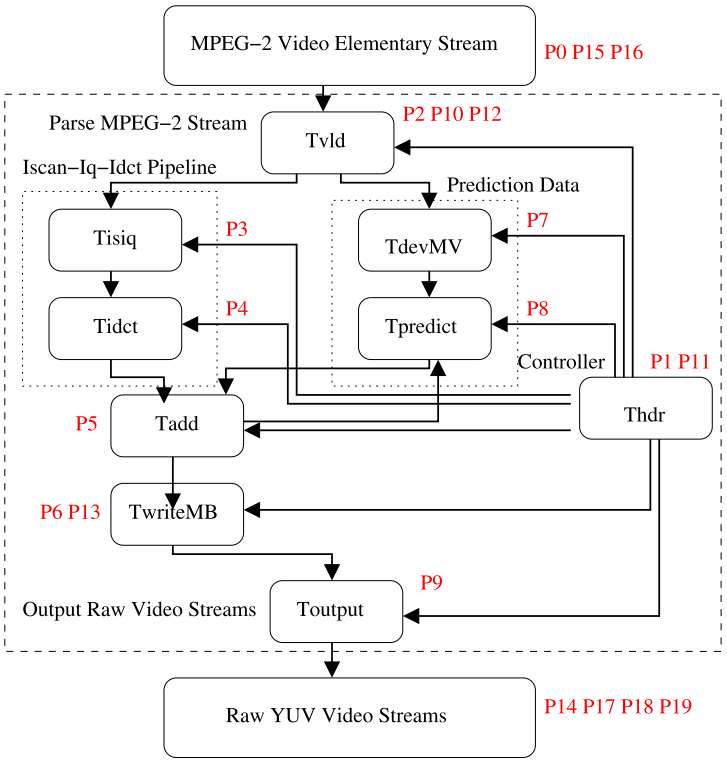

Fig. 6 System level diagram of the MPEG-2 decoder

blocking and the calling process is suspended to wait for the event that another process reads from the FIFO queue. Hence when the process reads from an empty FIFO queue, it waits for the sc_event which would be notified by the FIFO writer when writing to the FIFO queue. Similarly, when the process writes to a full FIFO queue, it waits for the sc_event which would be notified by the FIFO reader when reading from the FIFO queue. In our implementation of the MPEG-2 Decoder design, there are 19 thread processes and 126 events. A deadlock occurs if the FIFO queue sizes are too small. A process is blocked when writing to a full FIFO queue. Before it is resumed, the process is not capable to write to another FIFO queue or read from another FIFO queue to unblock other processes. If successive blocked process includes the process that read from the FIFO queue the first process writing for, a deadlock occurs and no process in the loop can continue execution. We experience such case when the one of the output FIFO queue size is too small in the Fig. 7. We have analyzed the run time of the MPEG-2 Decoder without deadlock. Although we limit all the FIFO queue sizes to be minimal and create extensive number of event waiting, simulation overhead keeps at a steady value of 7.3%. The run time simulation data has been shown in Table 2. The overhead for deadlock detection is relatively constant throughout the simulation lifetime. The overhead for each deadlock detection only depends on the complexity of the DSDG, which does not change dramatically. Vertices and edges are dynamically added and deleted during simulation, and the DSDG normally remains a sparse graph. Therefore, it only adds a small and constant overhead for each system-level synchronization statement.

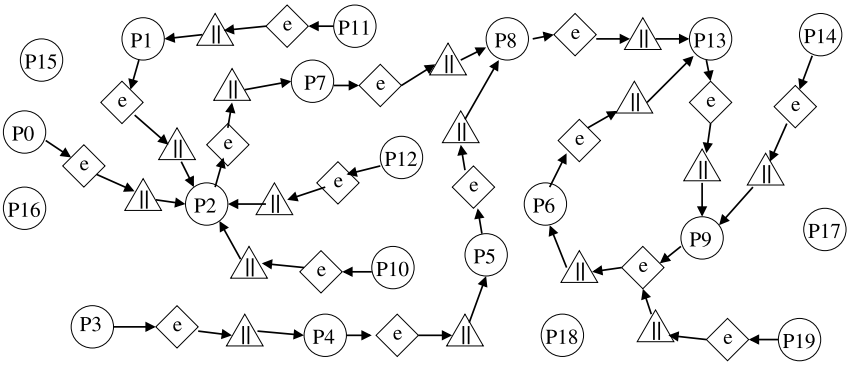

Fig. 7 A DSDG for MPEG-2 decoder

Table 2 Run time analysis of MPEG-2 decoder

#Frames	#Steps	Without deadlock	With deadlock	Overhead
0	37656	0.862 sec	0.924 sec	7.19%
1	52195	1.440 sec	1.575 sec	9.37%
2	73406	2.025 sec	2.178 sec	7.55%
3	79136	2.339 sec	2.518 sec	7.65%
4	87827	2.719 sec	2.914 sec	7.17%
5	92435	3.062 sec	3.302 sec	7.83%
6	96435	3.170 sec	3.410 sec	7.57%
7	98255	3.234 sec	3.466 sec	7.17%

4.2 Metropolis

Metropolis [22] is a system-level design framework for modern embedded systems. In the modeling language of Metropolis, Metropolis Meta-Model (MMM), a design is specified as asynchronous processes with communication specified with media and with its overall behavior limited by the synchronization constructs: function-architecture mappings, await statements, interface function calls, constraints, and schedulers. The function and abstract architecture of a system are specified separately and correlated by the synchronization of the functional events with architectural events (mapping). To limit the behavior of processes, designers can put high level LTL (Linear Temporal Logic) [23] or LOC (Logic of Constraints) [24] constraints on the system specification without giving any specific scheduling algorithm, and leave the implementation to the lower levels of abstraction. Designers can also write their own schedulers in architecture models at a high abstraction level, which are called quantity managers in Metropolis. The high flexibility of the design platform allows designers to use different modeling constructs freely in a system design. Without a platform-supported systematic analysis mechanism, this flexibility can lead to vulnerability to design errors that may cause deadlocks.

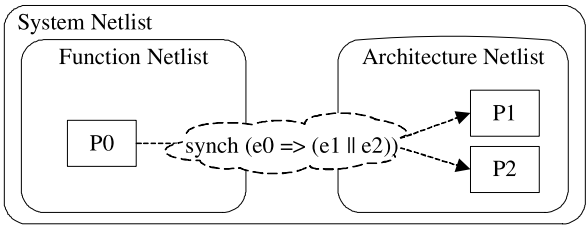

Fig. 8 An example of *synch* constraint

4.2.1 Synchronization language

The modeling constructs for synchronization in MMM include synch constraints, await statements, interface functions, quantity managers and LTL and LOC constraints. Most of these synchronization constructs are not unique to MMM, and their counterparts are also used in other concurrent modeling languages.

In Metropolis, the system function and the architecture are modeled as separate networks of processes communicating through media. In a functional network, functional processes run concurrently and communicate with each other through media. In an architectural network, computing and storage resources are modeled with media. Services that the architecture can provide are modeled with processes that are called *mapping processes*. A function model is mapped to an architecture model as the events of functional processes and mapping processes are synchronized with *synch* constraints. Designers are allowed to implement particular schedulers as *quantity managers* to manage architectural resources and services in an architecture model. Quantity managers are basically scheduling media that implement a particular set of functions that can be invoked by processes to issue service requests. An architectural mapping process may be suspended by a quantity manager if it requests resources (quantities in Metropolis terminology) from it. The corresponding functional processes that are mapped to the mapping process can then be blocked through *synch* constraints.

A *synch* constraint is an alternative of a rendezvous used in the concurrent programming [25, 26]. It can specify that two events in two different processes must occur at the same time. If only one of the two events can be scheduled to occur, the process containing the event has to be blocked until the other event can occur also. A *synch* can also require that an event cannot occur until any of the other events occur. The execution of a process has to be blocked at a certain event until all the *synch* constraints containing the event are satisfied. For example, assume functional process p_0 and mapping processes p_1 and p_2 have events e_0, e_1 and e_2, respectively, and are synchronized by a *synch* constraint $synch(e_0 => e_1||e_2)$, which requires that e_0 cannot occur until e_1 or e_2 occurs. This scenario may denote that a functional process cannot run until there are free computation resources in the architecture. The execution of p_0 may be blocked by either p_1 or p_2, as illustrated in Fig. 8.

An *await* statement is used to establish mutually exclusive sections and to synchronize processes. It contains one or more statements called *critical sections*, each controlled by a triple (*guard*; *testlist*; setlist). The *guard* can be any Boolean expression, and the *testlist* and *setlist* denote sets of interfaces, which essentially work as integer semaphores that can be incremented or decremented. A critical section is said to be *enabled* if its *guard* is evaluated to true and none of the interfaces in the *testlist* has been set by other processes in the system. A critical section may start executing only if it is enabled. While the critical section is being executed, the "semaphores" specified in the *setlist* are incremented and can block other processes that require the semaphores. The interface function calls are also prevented if the interface is set by an *await*. If no critical section is enabled, the execution blocks. If more

than one critical section are enabled, the choice is non-deterministic. For example, an *await* statement has two critical sections:

await {
  (*foo*(); $intf_{00}$; $intf_{01}$) {$critical\_section_0$;}
  (*true*; $intf_{10}$, $intf_{11}$; $intf_{10}$, $intf_{11}$) {$critical\_section_1$;}}

The first critical section is enabled only if guard *foo*() is evaluated to true and $intf_{00}$ is not set by other *awaits*. If a process enters this critical section, $intf_{01}$ will be set . The second critical section is enabled only if none of interfaces $intf_{10}$ and $intf_{11}$ is set by other processes. If a process enters this critical section, $intf_{10}$ and $intf_{11}$ will be set by the process. Note that an interface can be set by multiple processes at a given time and must be unset by all of them to be released.

Designers can also add general LTL and LOC constraints to a system to further restrict the behaviors of the system. We do not present these constraints directly in the paper since their specification semantics are not for execution and it is up to the simulator to make sure that the execution is consistent with the constraints.

4.2.2 Deadlock in Metropolis

Given the constructs considered in MMM, only the following situations may block the execution of a running process:

1. A process has to wait for synchronization from other functional or architectural processes as required by one or more *synch* constraints.
2. A process cannot execute an interface function due to the fact that the interface is included in the setlist of a critical section being executed in another process's *await* statement.
3. A process is blocked at an *await* statement due to the unsatisfaction of all its guard/testlist conditions.
4. A mapping process is suspended by a quantity manager when it is requesting some quantity from it but cannot be satisfied.

The interaction of these synchronization constructs can be quite complicated. A deadlock exists if and only if there exist dependency loops among the processes in a system. We will identify and analyze the deadlock situation and report the processes and the media to which they are connected. The synchronization dependency analysis is useful to provide a guide to the designers to help isolate the problem.

4.2.3 Case study: picture-in-picture video processing system

In this section, we use a real design of a complex function model for Picture-in-Picture video processing and a high level model of function-architecture mapping to demonstrate the usefulness and effectiveness of our deadlock analysis approach for system-level designs.

Figure 9 shows a picture-in-picture (PiP) video processing design. TS DEMUX demultiplexes the single input transport stream (TS) into multiple packetized elementary streams (PES). PES PARSER parses the packetized elementary streams to obtain MPEG video streams. Under the control of the user (USRCONTROL), decoded video streams can either be resized (RESIZE) or directly feed to JUGGLER that combines the images to produce the picture-in-picture videos. RESIZE is the major component of PiP that computes and adjusts the size of MPEG video frames according to user inputs. It consists of about 9,000 lines of

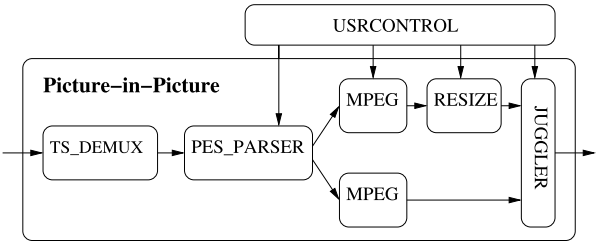

Fig. 9 Picture-in-picture design

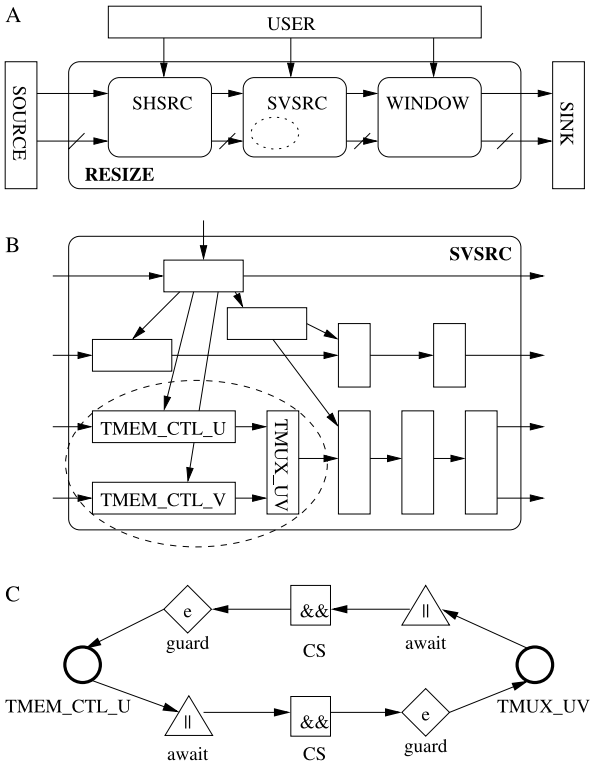

Fig. 10 The RESIZE unit and its synchronization dependencies

Metropolis Meta-Model source code and contains 22 concurrent processes and more than 300 media.

The video frames and control signals are passed between processes through around 80 communication channels specified with media. The communication channels are modeled at the task transition level (TTL) with bounded first-in-first-out (FIFO) buffers [27]. The mutual exclusion and boundary checking of the bounded FIFO buffer is guaranteed by a central protocol. To simulate the RESIZE unit, three additional processes are used to mimic user inputs (USER), send MPEG video streams to the unit (SOURCE) and absorb the data from it (SINK) as shown in Fig. 10A.

In the simulation with our runtime deadlock monitoring mechanism enabled, a deadlock is reported immediately after TMUX_UV and TMEM_CTL_U block each other through two await statements and their synchronization dependencies are captured in the DSDG as shown in Fig. 10C. As it turns out, there is a design error in process TMUX_UV,

Table 3 Simulation and analysis summary for both case studies

Example	RESIZE unit	Mapping model
Code size	9000 lines	5900 lines
Processes	22	8
Media	300+	16
Deadlocked processes	2	5
Time to catch deadlock	2 min	<1 min

which fails to read all the data sent by TMEM_CTL_U. The data in the bounded buffer of the channel between the two processes accumulates until the buffer becomes full. Then a deadlock occurs where TMEM_CTL_U is blocked waiting for the buffer space to be released by TMUX_UV while TMUX_UV is also blocked waiting for reading signals from TMEM_CTL_U. The designers can now focus on the two processes and the communication channels between them to identify and correct those design errors. A solution is to modify process TMUX_UV and make it absorb all the data from its input channels even if not all the data is useful. We observe that, without the deadlock detection mechanism, the simulation will continue and the regular simulation trace won't show any apparent sign of deadlock until most of the processes in the system are eventually blocked. By that time, the simulation trace is long and a large number of processes are blocked. Our approach automatically catch the deadlock as it first occurs. Designers can then focus on solving the deadlock without complicating themselves by the consequences of the deadlock. The simulation and analysis results are summarized in Table 3.

In the platform-based design, the mapping is the key procedure that correlates the function to the architecture. In this design example (as shown in Fig. 11A), two source processes (S1 and S2) write the data into two independent channels. A separate process (Join) then reads data items from both channels, manipulates them, and then sends the result data to another process (Sink) through another channel. In the abstract architecture model, there are two CPU/RTOS units, a bus unit, a memory unit and a quantity manager (i.e. scheduler) for each architectural unit. A CPU unit can be shared among several software tasks that may request services from it. When more than one service request is issued to a CPU, arbitration is needed. The mapping procedure synchronizes the processes in the function model and the mapping processes (representing software tasks) in the architecture model. In this example (as shown in Fig. 11A), functional processes S1 and S2 are mapped to mapping processes SwTask1 and SwTask2, respectively, which are associated to CPU1 and the other two processes are mapped to CPU2. The CPU quantity managers implement a non-preemptive static-priority dynamic scheduling policy. The two CPUs are connected to the bus and the bus is connected to the memory unit.

Our deadlock detection mechanism reports a deadlock within one minute of simulation. Due to the boundedness of the channels between processes, process S1 cannot complete a task of writing data before Join reads from and releases the channel buffer. Therefore, with the current CPU scheduling policy, the deadlock occurs when S1 obtains the CPU service but cannot complete a writing task while Join is still waiting for data from S2 who cannot get CPU service. The deadlock situation involves five processes, two await statements, two *synch* constraints and a quantity manager as shown in Fig. 11B. This analysis also suggests several possible deadlock resolutions. The deadlock can be resolved by making the channel buffer large enough to store all the data from a single writing task, increasing the number of CPUs, or changing the CPU scheduling policy. We also observe that such deadlocks only occur in the mapped design and are not inherent in the function specification or in

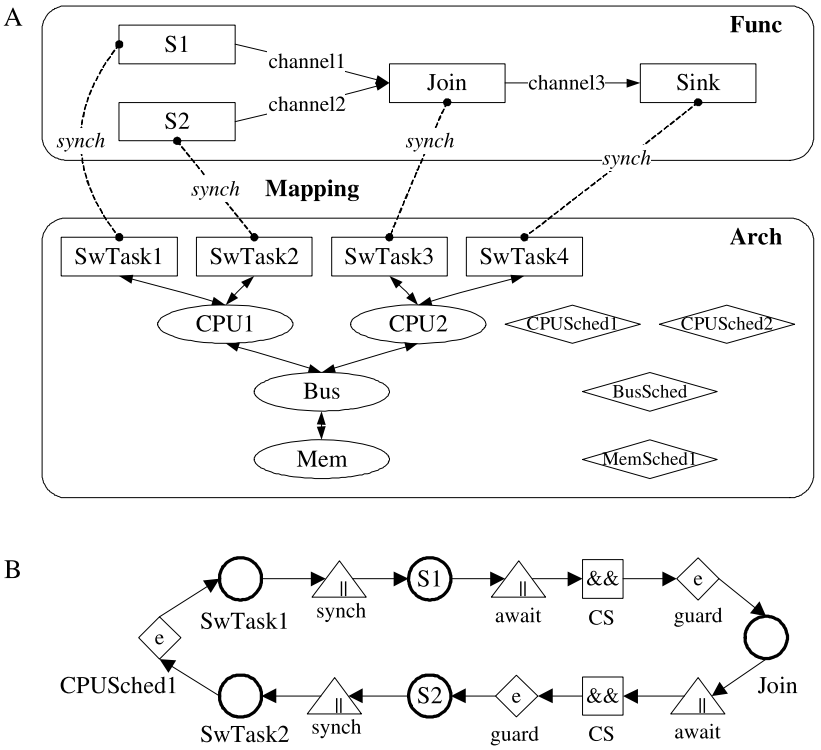

Fig. 11 A mapping model and its synchronization dependencies

the architecture model. The simulation and analysis results for this mapping model are also listed in Table 3.

5 Conclusion

In this paper, we study the deadlock problem in system-level designs that include complex synchronization constructs and functionality. We present our deadlock detection approach with a data structure called the dynamic synchronization dependency graph and an associated deadlock detection algorithm. We use two publicly available system-level design tools, Metropolis and SystemC, and two real world design examples to demonstrate the usefulness and effectiveness of our deadlock detection approach for system-level designs. We believe that the dynamic synchronization dependency graph can be used to not only detect deadlock situations but also help search for livelock or starvation problems. We are currently working on combining our simulation-based deadlock detection mechanism and existing formal verification techniques to search for more subtle problems such as livelock and starvation in system level designs.

Algorithm 5 Deadlock Avoidance

```
1: procedure deadlockAvoid(L_i)
2: Find all the e_s in cycle L_i;
3: Let T = Set of all the resolution tags corresponding to all e_s;
4: sort T in increasing order and let n = the cardinality of T;
5: for k = n − 1 to 0 do
6:    if All the process vertices at resolution stage T_k have been marked being executed
      first then
7:       continue;
8:    else
9:       reset all the deadlocked processes to the resolution stage T_k;
10:      choose a process which has not been executed first at stage T_k;
11:      Mark the first executed process of the stage T_k;
12:      start the simulation from the sc_time_stamp of T_k;
13:   end if
14: end for
15: Return Deadlock cannot be avoided;
16: end procedure
```

6 Future direction

Due to the non-deterministic nature of the system-level design, different non-deterministic resolutions can lead to different simulation traces, all equally valid. A simulator naturally can only simulate one such resolution, so it is possible to choose a resolution that is more likely/unlikely to find a deadlock. To search for deadlock, a simple heuristic may choose a resolution that will lead to more dependency. In our simulation environments, we can easily have the simulator "test run" different resolution for a few cycles and then choose the most "busy" one. We are currently investigate the efficiency and usefulness of such an algorithm.

Alternatively, the designers may be willing to overlook, for now, the resolution that will cause a deadlock and concentrate on the normal functioning of the system. Further synthesis step or manual manipulation may then make the deadlock due to a particular non-determinism resolution irrelevant. At each resolution point, we take a snapshot (the process name, the model name, the suspended stage, and the blocking information) of all the processes that are either suspended or ready to run and store in an array of queues. We tag a serial number to this resolution.

In Algorithm 5, we find all the resolution stages that introduce dependencies in the DSDG. We sort these resolutions as per their tag numbers and reset our processes to the data present in queue corresponding to the last resolution. We find an alternative execution order at this stage and start simulation from there. If all the possible execution orders of this stage hit the deadlock then we try moving one stage back. And this continues until either we find a way to avoid the deadlock or we fail avoiding the deadlock considering all the possible execution orders of all the identified resolution stages. The performance of the algorithm is heuristic in nature and depends on the number of the resolution stages we consider to look back. Currently our implementation looks back one resolution stage. For evaluation purpose we designed a small example as shown in Fig. 12. We have three processes interdependent on each other by blocking dependencies. The process chosen at each resolution stage has been shown in bold. We hit a deadlock just after resolution stage 4. Process a waits on c, process c waits on b and process b waits on process a. We apply our heuristic of rolling

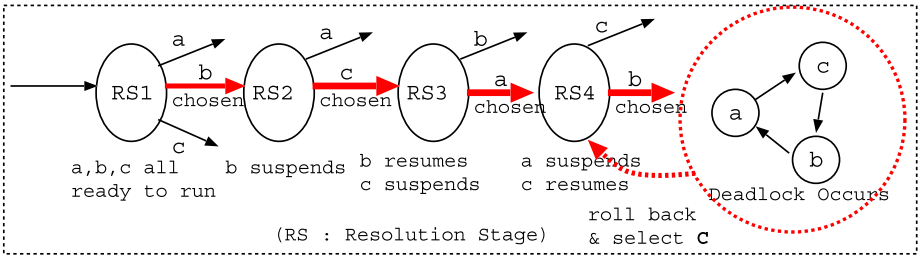

Fig. 12 An example showing avoiding deadlock

back to the last stage and choose process *c* to execute before process *b* which avoids the deadlock and simulation continues.

We are also working on other communication error such as starvation and livelocks. Starvation is simply defined as processes that could run but are not allowed to run by the scheduler for a "long" period of time. If a scheduling process is modeled by the designers, then it is up to them to build in a starvation detection or make their scheduling starvation free by using the likes of watchdog timers. We are working on starvation detection example where processes are starved by system-level simulator scheduler. Obviously, how long is "long" will be a user-defined parameter. Livelocks can be defined as processes busy working but no progress are made. We are looking at detection of livelocks through two separate methods. First is to look at the system states that "keeps repeating". The second is to look at the dependency patterns that "keeps repeating". We are currently gathering examples of starvation and livelocks from industrial design to make our study more relevant.

References

1. Coffman EG, Elphick M, Shoshani A (1971) System deadlocks. ACM Comput Surveys 3(2):67–78
2. Sfinghal M (1989) Deadlock detection in distributed systems. IEEE Comput 22(11):37–48
3. Knapp E (1987) Deadlock detection in distributed databases. ACM Comput Surv 19(4):303–328
4. Peterson JL, Silbershatz A (1983) Operating system concepts. Addison-Wesley, Reading
5. Coffman MEEG, Shoshani A (1971) System deadlocks. ACM Comput Surv 3(2):67–78
6. Sfinghal M (1989) Deadlock detection in distributed systems. IEEE Comput 22(11):37–48
7. Knapp E (1987) Deadlock detection in distributed databases. ACM Comput Surv 19(4):303–328
8. Mitchell DP, Merritt MJ (1984) A distributed algorithm for deadlock detection and resolution. In: ACM symposium on principles of distributed computing, 1984, pp 282–284
9. Krishnamurthi M, Basavatia A, Thallikar S (1994) Deadlock detection and resolution in simulation models. In: 26th Conference on winter simulation, 1994
10. Habermann AN (1969) Prevention of system deadlocks. Commun ACM 12(7):373–377
11. Godefroid P, Pirottin D (1993) Refining dependencies improves partial-order verification methods. In: Proceedings of the 5th conference on computer aided verification. Lecture notes in computer science, vol 697. Springer, Berlin, pp 438–449
12. McMillan K (1993) Symbolic model checking. Kluwer Academic, Dordrecht
13. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–258
14. Jain K, Hajiaghayi MT, Talwar K (2005) The generalized deadlock resolution problem. In: International Colloquium, ICALP, July 2005
15. Krishnamurthi M, Basavatia A, Thallikar S (1994) Deadlock detection and resolution in simulation models. In: Proceedings of the 26th conference on winter simulation. Society for Computer Simulation International, San Diego, pp 708–715.

16. Functional specification for systemC2.0, update for systemC2.0.1 version 2.0-q. April 2002
17. Black DC, Donovan J (2004) SystemC: From the ground up. Kluwer Academic, Dordrecht
18. Bhasker J (2002) A SystemC primer. Star Galaxy Publishing, Allentown
19. Grotker T, Liao S, Martin G, Swan S (2002) System design with systemC. Kluwer Academic, Dordrecht
20. van der Wolf P, Lieverse P, Goel M, Hei DL, Vissers K (1999) An MPEG-2 decoder case study as a driver for a system level design methodology. In: Proceedings of international workshop on hardware/software codesign, May 1999
21. Kahn G (1974) The semantics of a simple language for parallel programming. In: IFIP Congress, 1974
22. Balarin F, Watanabe Y, Hsieh H, Lavagno L, Passerone C, Sangiovanni-Vincentelli A (2003) Metropolis: an integrated electronic system design environment. IEEE Comput 36(4):45–52
23. Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems: specification. Springer, Berlin
24. Balarin F, Watanabe Y, Burch J, Lavagno L, Passerone R, Sangiovanni-Vincentelli A (2001) Constraints specification at higher levels of abstraction. In: Proceedings of international workshop on high level design validation and test, Nov. 2001
25. Hoare CAR (1978) Communicating sequential processes. Commun ACM 21(8):666–677
26. Charlesworth A (1987) The multiway rendezvous. ACM Trans Program Lang Syst 9(3):350–366
27. Gangwal O, Nieuwland A, Lippens P (2001) A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. In: Proceedings of international symposium on system synthesis, Oct. 2001